自動的等価性差分の抽出による SSA コンパイラ最適化器の正しさの検証

Verification of Compiler Optimization on SSA form by Finding Value Equality Difference

Fang L.　佐々 政孝 †

† 東京工業大学情報理工学研究科

Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

{fang3, sassa}@is.titech.ac.jp

最適化はコンパイラの重要なフェーズである．コンパイラ最適化器は，入力プログラムの振舞いを変えてはいけない．コンパイラは複雑なソフトであるため，一般に，コンパイラの誤りの原因を突き止めることが難しい．本論文では，最適化前後の等価性の差分の自動抽出に基づいて SSA 形式上のコンパイラ最適化器の正しさを検証する手法を提案する．最適化前後のプログラムを比較照合し，最適化による変形を抽出し，それがプログラムの意味を変えない等価変形であるかどうかを検査する．等価変形の性質は時相論理で記述し，正しさの検証はモデル検査による．本手法により COINS コンパイラの最適化器の誤りや曖昧な変形をいくつか発見した．検査器は効率的であり，良い効果があり，使いやすい，などの利点がある．

Optimizers are very important phases of compilers. It is essential that the compiler optimizer is implemented without changing the semantics of a program. How to guarantee the correctness of optimization of real programs is still an unsettled problem. In this paper, we propose a technique for validating whether the optimization transformation of the program have been done correctly or not by verifying whether the optimization transformations of program are equivalent transformations. We compare the program before and after optimization and find the changed points. The properties that the points have to satisfy are described by temporal logic formulas, and we check whether the temporal logic fomulas are satisfied at these points using model checking. We applied this technique to the complex real optimizers of COINS, and it works with great effectiveness and efficiency. We have found some ambiguous and incorrect transformations and hope to find more bugs in the COINS compiler.

1　はじめに

1.1　背景

コンパイラのプログラム最適化は重要な技術であり，近年盛んに研究されている．最適化は目的プログラムの時間および空間効率を向上させるプログラム変換を行う．

コンパイラ最適化器は，入力プログラムの振舞いを変えてはいけない．入力プログラムの振舞いを変えてしまうような最適化器は，仕様もしくは実装に誤りを含んでいる．

コンパイラは複雑なソフトであるため，一般に，アルゴリズムの設計や実装の段階など様々な箇所で，プログラムの意味を変えてしまうような誤りが混入しやすい．最適化が正常に終了したように見えても，最適化された目的プログラムは意図しない動作をするかもしれない．さらに，違う挙動を示しても，その原因がどこで混入したものなのか見極めにくい．

このような背景から，最適化器のバグがないことを保証するための技術は非常に重要である．

コンパイラ最適化器の信頼性を向上させる既存の研究として，次のようなものがある．

1. 最適化器そのものが正しいことを検証する．検証された最適化器は，任意のプログラムについて，その振舞いを変えることなく最適化できる．
2. 最適化の実行中に，最適化されるプログラム上で最適化器の振る舞いを確かめる．
3. 最適化の実行後に，プログラムの意味が変わら

ないような変形であったことを検査する．

4. 最適化の実行後に，最適化前後のプログラムの実行を比較照合し，最適化前と最適化後のプログラムについて、両者の実行途中の変数の値を比べることで、意図とは異なる振舞いの変数を探し出す。

1 の研究には，Lacey らの研究 [11][12] や Lerner らの研究 [13][14] などがある．これらの研究は証明できるドメイン言語を定義することによって，理論上の厳密性を保証するが，簡単な最適化しか扱えなく、実用性に欠ける点がある．

2 の研究には，佐原らの研究 [21] がある．この研究は最適化器の振舞いを検査の対象とし，最適化器を実行しながら，最適化のアルゴリズムと関係ある箇所にマークを付け，それらのマーク間の関係が時相論理式を満たすかどうかを検査する．最適化器を拡張する必要があるため，処理が煩雑で効率が良くない．また，最適化器と最適化器のアルゴリズムに依存するため，ユーザが最適化器や最適化のアルゴリズムを理解する必要があり，定式化が多数にわたったり，関係が複雑な場合は対応できない場合もあるなどの欠点がある．

3 の研究には，Rinard らの研究 [18] や Necula の研究 [17] などがある．Rinard らの研究は，プログラム変形の正当性を厳密に示せるが，実際に適用する際にどの程度実用的かは明らかではない．Necula の研究は最適化前後のプログラムの意味が等しいことを，記号的に推測し評価することで検査する手法を提案した [17]．この検査手法は，原理的には最適化に依存しない利点がある．しかし，この研究は厳密なプログラムの意味の保存を保証できず，推測が失敗する場合があり，複雑な最適化に対しては精度が低いと思われる．また，バグを発見した際，バグの場所を確定できない，などの欠点がある．

4 の研究には，佐々らの研究 [19] や Jaramillo らの研究 [10] などがある．この研究は最適化前後のプログラムを交互に実行して行き，対応する変数に対する値が一致しているかどうかを検証する手法である．しかし，トレースデータが大きすぎ，検証時間が長い，理論上の信憑性が弱いなどの欠点がある．

1.2 概要

本論文は 3 の研究に属する．本手法は，最適化前後のプログラムの差異を自動的に抽出し，最適化による変形が意味的に同値性を持つかどうかを検査する点に特徴がある．

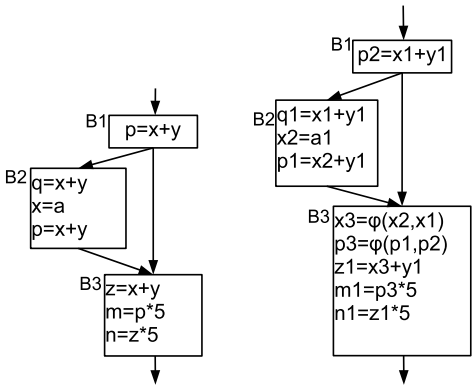
本手法の流れは次のようになる．静的単一代入形式の中間表現を対象とする．まず，最適化前後のプログラムを解析し，$def-use$ などのプログラムの性質を解析し，最適化による変形箇所を抽出する．次はモデル検査を行い，最適化実行後に変形箇所が同値変形であったかどうかを調べ，間違った変形を報告する．その前に，変形箇所がプログラムの意味を保つために満たすべき性質を時相論理 CTL-FV[12] で記述しておく．

我々の手法による利点は以下の通りである．

- 既存の最適化器に手を入れることなく適用できる．
- 最適化器や最適化のアルゴリズムに依存しない．そのため，
  - ユーザが最適化や最適化のアルゴリズムを理解しなくてよく，使いやすい
  - 定式化しやすい
  - 高速で効率が良い
- バグを発見した際に原因の特定が容易である．
- 条件付き定数伝播 [23] や質問伝播に基づく大域値番号付けと部分冗長除去 [20] といった非常に複雑な最適化器の検証ができる．
- 効果が高くて実用性がある．

本手法を用いて，COINS コンパイラ [6] に実装されている最適化器に対して，本手法を適用して検査を行った．まだ実験データをまとめるに至っていないが，多くの誤りや曖昧な変形が発見できる傾向にある．

本論文の構成は次の通りである．2 節はプログラムの SSA 形式及び SSA 形式上での最適化について説明する．3 節は検証に使う時相論理について述べる．提案手法の詳細は 4 節で，5 節では最適化による変形が満たすべき性質を記述する定式化について簡単に言及する．手法の有用性の考察は 6 節で述べる．



(a) 元のプログラム　　(b) SSA 形式

図 1　SSA 形式への変換の例

7 節，8 節，9 節はそれぞれ関連研究，まとめ及び将来課題について述べる．

2　プログラムの静的単一代入形式 (SSA 形式) と SSA 形式上での最適化

この節では，プログラムの SSA 形式，SSA 形式上の最適化について述べる．

2.1　静的単一代入形式

静的単一代入形式（SSA 形式）とは，変数の定義がプログラムの字面上唯一となるようにしたプログラムの表現形式である [16, 2]．SSA 形式では，変数の使用に到達する定義が一意に決定でき，プログラムの最適化に有利な形式といわれている．

図 1 は，プログラムの SSA 形式への変換の例である．図 1(a) のプログラムを SSA 形式に変換すると，図 1(b) のようになる．

通常形式のプログラムを SSA 形式に変換するには，

- 変数の名前の付け換え
- ϕ 関数の挿入

を行う．ϕ **関数**とは，元のプログラムで同じ変数の定義が合流するところに挿入される仮想的関数である．図 1(b) の B3 にある ϕ(x2, x1) などが ϕ 関数である．この ϕ 関数は，「B2 から来た場合は x2 を，B1 から来た場合には x1 を返す」関数である．

これを少し説明すると，図 1(b) の B3 において，x の値は，B2 から来た場合は B2 内で定義された x2 の値，B1 から来た場合は B1 内で有効な x1 となる．B3 の入口に ϕ 関数による代入文 x3 = ϕ(x2, x1) を挿入することで，B3 での x の値は x3 と決定することができる．

SSA 形式は，次の有用な性質を持つ．

- 各変数の使用には，唯一の定義が到達する．
- 制御フローグラフ上のノード n で，変数 v の異なる定義が合流するとき，ϕ **関数**を n の先頭に挿入することで，到達する v の値を区別する．

2.2　SSA 形式上での最適化

プログラムの最適化とは，コンパイラが，プログラムの実行速度やサイズなどの性質を改善する目的で行うプログラムの変形のことを指す．最適化は，プログラムの性質を**解析**し，その結果に基づき**変形**するという方法が一般的である．

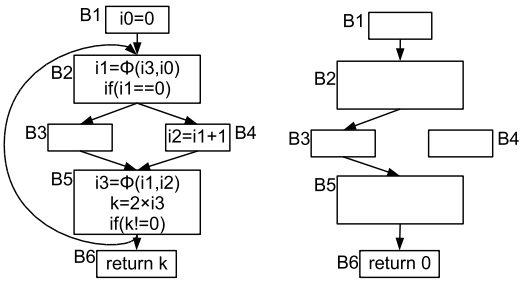
プログラムの性質を解析する手法は色々あるが，制御フローグラフ上のデータフロー方程式を解くことにより行うのが一般的である．しかし，分岐を考慮した定数伝播 [23] のような，制御フローグラフ上のデータフロー方程式を解いたのでは解が得られず，制御フローグラフを辿り，値がどうなるかを調べる**抽象実行（abstract interpretation）**を行う最適化もある．図 2 は条件分岐を考慮した定数伝播の例である．

最適化器は，最適化前後でプログラムが意味的に変わっていないことを保証しないといけない．プログラムが意味的に変わっていないことを，**プログラムの意味が保存される**という．プログラムの意味が保存されないような最適化は，正しくない最適化である．

最適化の正しさとしては他に，その最適化による変形が確かにプログラムの効率を向上させているという性質を満たすということが挙げられる．しかし，本当に最適かどうかは難しい問題で，一般に示すことができない性質である．

そこで以下，最適化が正しく実行されたとは，少なくとも最適化されたプログラムの意味が保存された

である．



(a) 元のプログラム　(b) 最適化された後のプログラム

図 2　条件付き定数伝播の例

ことをいう．

3　時相論理

本手法の検査には，時相論理 CTL-FV によるモデル検査を用いる．この節では，まず検査対象であるモデルについて述べ，その後 CTL-FV の構文や意味について述べる．

3.1　状態遷移モデル

CTL-FV によるモデル検査を行うためには，プログラムを状態遷移モデルとして形式的に表現する必要がある．プログラムの最適化は制御フローグラフ上で行われるので，本手法で用いる状態遷移モデルは制御フローグラフ [1, 2, 16, 22] とその上に行った変化を基にするのが自然である．モデル生成の詳細は 4.2.3 節を参照されたい．

3.2　CTL-FV

本手法で用いる時相論理は，Lacey らが提案した **CTL-FV** である [12]．これは，分岐時間時相論理の一種である CTL[4, 5] を基にした論理で，次のような特徴がある．

- 逆向きの経路に関する限量子 $\overleftarrow{E}$，$\overleftarrow{A}$ を扱える．
- 原始命題を自由変数を引数にとる述語に一般化している．

直観的には，$\overleftarrow{E}$，$\overleftarrow{A}$ は通常の向きの経路限量子 E，A をそのまま逆向きの経路限量子としたものである．

我々が CTL-FV を採用した理由は次の 2 つである．

- 制御フローやデータフローの性質の記述に適している．
- 効率よくモデル検査を行うことができる．

■CTL-FV の構文　CTL-FV の構文規則を表 1 に示す．ϕ は状態に関する式（**状態式**）を導出する非終端記号，ψ は経路に関する式（**経路式**）を導出する非終端記号であり，α は自由変数を引数とする述語である．

$\phi ::= true$	$\phi ::= E\ \psi$	$\psi ::= X\ \phi$
$\phi ::= false$	$\phi ::= A\ \psi$	$\psi ::= \phi\ U\ \phi$
$\phi ::= \alpha$	$\phi ::= \overleftarrow{E}\ \psi$	$\psi ::= \phi\ W\ \phi$
$\phi ::= \neg\phi$	$\phi ::= \overleftarrow{A}\ \psi$	
$\phi ::= \phi \wedge \phi$		

表 1　CTL-FV の構文規則

■CTL-FV の意味論　モデル M 上の状態 n で状態式 ϕ が成立することを，$M, n \models \phi$ と表す．また，経路 p で経路式 ψ が成立することを，$M, p \models \psi$ と表す．どちらも，M が自明であるときには省略して単に $n \models \phi$，$p \models \psi$ と表す．

CTL-FV の制御フローモデル上での意味の定義を表 2 に示す[*1]．なお，以下では制御フローグラフのノードは 1 つの文からなるとする．

N，L を制御フローにおけるノードと有向辺の集合，L をノードで成り立つ原始命題の集合を与える写像とするとき，モデル $M = (N, E, L)$ における $L(n)$ の定義はプログラムの性質及び最適化による変化に従って表 3 に示すようになる．

4　提案手法

この節は提案手法の概要及び提案手法の実現手順の詳細について述べる．

[*1] 厳密には，式中の自由変数をプログラム中のシンボルで全て束縛した式について意味は定義される．

状態式 $(n \rightarrow^{\circ} n_1$ means $n_1 \rightarrow n\ )$

$n \models \mathit{true}$	iff	true
$n \models \mathit{false}$	iff	false
$n \models \alpha$	iff	$\alpha \in L(n)$
$n \models \neg\phi$	iff	not $n \models \phi$
$n \models \phi_1 \wedge \phi_2$	iff	$n \models \phi_1$ and $n \models \phi_2$
$n \models E\psi$	iff	$\exists p = n \rightarrow n_1 ...\ ,\ p \models \psi$
$n \models A\psi$	iff	$\forall p = n \rightarrow n_1 ...\ ,\ p \models \psi$
$n \models \overleftarrow{E}\psi$	iff	$\exists p = n \rightarrow^{\circ} n_1 ...\ ,\ p \models \psi$
$n \models \overleftarrow{A}\psi$	iff	$\forall p = n \rightarrow^{\circ} n_1 ...\ ,\ p \models \psi$

経路式 $(p = n_0 \rightarrow' n_1 ...$ in which $\rightarrow'$ is $\rightarrow$ or $\rightarrow^{\circ})$

$p \models X\phi$	iff	n_1 exists and $n_1 \models \phi$
$p \models \phi_1 U \phi_2$	iff	$\exists i \geq 0\ [\ n_i \models \phi_2$ and $\forall j\ [\ 0 \leq j < i$ implies $n_j \models \phi_1\ ]]$
$p \models \phi_1 W \phi_2$	iff	$(p \models \phi_1 U \phi_2)$ or $(\forall k \geq 0\ [\ n_k \models \phi_1\ ])$

表 2　CTL-FV の意味定義

$L(n) =$

- $\cup$ { use(X): 変数 X は n で使用される }
- $\cup$ { def(X): 変数 X は n で定義される }
- $\cup$ { $trans$(E): 式 E は n で変更されない，つまり E 中の変数は n で定義されない }
- $\cup$ { rm_def(X):変数 X は n で定義されるが，最適化によって削除される }
- $\cup$ { rm_use(X): 変数 X は n で使用されるが，最適化によって削除される }
- $\cup$ { rpl_var(X1 $\rightarrow$ X2) : n で変数 X1 が最適化によって変数 X2 に書換えられる }
- $\cup$ { rpl_exp(E1 $\rightarrow$ E2) : n で式 E1 が最適化によって式 E2 に書換えられる }
- $\cup$ { rm_branch(e): n で辺 e が削除できる }
- $\cup$ { ins_def(X):n は X に対する定義文が挿入されたノード }

表 3　$L(n)$ の定義

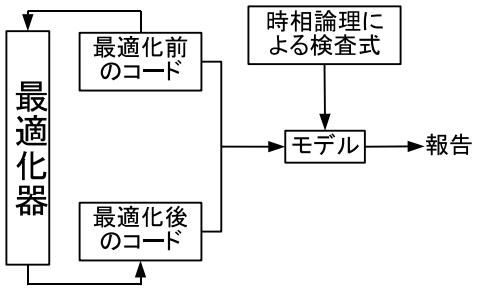


図 3　提案手法の概要

4.1　提案手法の概要

プログラムの意味を変えていない変形は**等価変形**という．本論文では，最適化が行うプログラム変形が，等価変形であるかどうかを時相論理によって検査する手法を提案する．図 3 は，提案する手法の概要を表す．

この方法は，現実のプログラムを比較するにはややこしいが，従来研究のように，検証できるように最適化器を作ったり最適化器を改造したりすることに比べて容易である．

最適化前後のプログラムを比較するには，変数の等価性を求める必要がある．変数の等価性を見つけるアルゴリズムには [3][16] など色々あるが，本手法は独自のやり方を用いている．この手法は定数伝播 [22][16] や定数の畳み込み [22][16] を含め，質問伝播に基づく大域値番号付けと部分冗長除去 [20] といった，非常に複雑な最適化を行った場合の変数の等価性も求められる．

モデルは単純に最適化前や最適化後のプログラムに基づいたものではなく，最適化前後の制御フローグラフを統合したものである．

4.2　提案手法の実現手順

提案手法は，次の 6 つのステップからなる．

■ 予備

1. 最適化の正しさの条件を検査仕様として記述しておく．

■ 最適化前後のプログラムを比較照合

2. 最適化前後のプログラムの変数の等価性を判定する．

3. 最適化前後のプログラムの命令文を比較照合する．
4. 最適化前後のプログラムの辺を比較照合する．

■ モデル検査

5. 最適化前後の制御フローグラフに基づいてモデルを生成する．
6. モデル検査し，結果を報告する．

各ステップの詳細は以降で順次説明していく．ここで特徴的なのは，この手法は最適化アルゴリズムを知らなくても適用できる汎用的なものである点である．

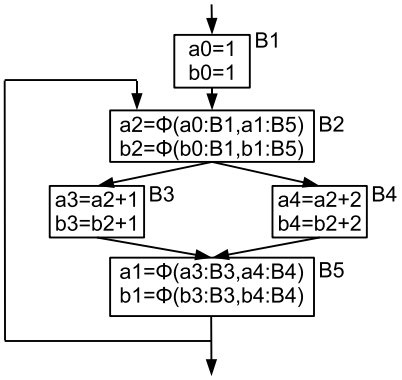


図 4　変数の等価性の例

4.2.1　予備

■検査仕様の記述　本手法ではまず，最適化による変形箇所がプログラムの意味を保存するために満たすべき検査仕様を，CTL-FV を用いてあらかじめ記述しておく．この記述を**検査仕様**ということにする．

例：不要命令文を除去する条件式：

$\neg(E\ (\neg\ def(x)\ U\ (use(x) \wedge \neg(rm\_use(x)))))$

この式は 3.1 節に定義した時相論理の意味論に基づいて次のように解釈する．変数 x が定義した後使用しないか、最適化による使用が削除されたかである．

検査は変形を対象としているため，最適化器や最適化器のアルゴリズムとは関係ない．従って，以下の利点が得られる．

- 仕様は最適化ごとに個別に記述する必要がない．
- ユーザが最適化アルゴリズムを理解しなくても定式化できる．
- 最適化のアルゴリズムが変わっても適用できる．
- 新しい最適化器を追加しても，検査器はそのまま使える．
- 定式化はプログラムの最適化の変形が満たすべき条件を記述するため，簡単に書ける．
- 定式の数が極めて少ない．

4.2.2　最適化前後のプログラムを比較照合

■変数の等価判定　2.1 節に述べたように，ここで扱う SSA 形式は，変数の定義がプログラムの字面上唯一となるようにしたプログラムの表現形式である．本研究では ϕ 関数や ϕ 関数によって代入される変数は，一般に値が実行経路に依存するので，**一意性がない**という．変数が実行経路に依存しないことを**一意性がある**という．

本手法は一意性のない変数について，変数の計算の振る舞いを表現できる計算木というものを求めることによって，同じ計算木を持つ変数が，等価計算していることがわかるようになっている．

値が同じ計算を**等価計算**という．等価計算によって定義された変数を**等価変数**という．等価変数の集合は**合同 (congruent)** という．

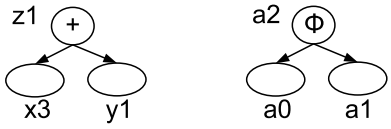
本手法は以下の手順で合同を求める．ここで図 1 の $z1, p3$ と図 4 の $a2, b2$ を例とし，ほかの等価変数の集合は同様に求められる．

- 一意性がある変数の代入は、代入文の左辺と右辺を合同という集合に入れ，合同に名前を付ける．右辺が定数同士による計算の場合，定数畳み込みを行い，計算結果も合同に入れる．例えば，図 4 の $a0 = 1;\ b0 = 1;$ に対して，合同 $\{a0, 1\}$, $b0, 1\}$ が求められ，合同の名前をそれぞれ $cong0$，$cong1$ とする．
- 一意性がない変数に対しては，図 1 と図 4 の SSA 形式のプログラムを例として説明する．計算の振舞いを表現できる計算木を以下の手順で計算する．
  - 代入文の右辺のオペレータを親ノードとし，オペランドを子ノードとし，計算木を生成する．図 1 の $z1$ に対して生成した木は図 5(a) である，図 4 の $a2$ に対して生成した木は図 5(b) である．

– 子ノードがさらにほかの計算木によって計算される場合, その子ノードをさらに展開する．例えば, 図 4 の $a2$ を展開すると図 6(a) のようになる．同様に $b2$ を展開すると，図 6(b) のようになる．
– ϕ 関数の場合，親ノードが空ではなく，かつ ϕ 関数ではない場合，ϕ 計算の分配律を適用し，親ノードを展開する．例として図 1 の $x3 = \phi(x2, x1)$ を $z1 = x3 + y1$ に分配すると，図 7(a) のようになる．同様に図 1 の $p3$ を展開すると，図 7(b) のようになる．
– 変数または定数がすでにいずれかの合同に属していた場合，変数はその合同の名前に入れ替え，展開を終了する．例えば図 6(a) のノード $a0$ を $cong0$ で置き換え，展開を終了する．
– すでに親ノードで一度展開したことがあるようなノードがあった場合も，展開した時点を参照するように記号を付け，展開を終了する．例えば図 6(a) の葉のノード $a2$ を $reference\ a2$（ノード $a2$ を参照する）で置き換え，展開を終了する．
– 以上の処理を行った後，計算木を式の形に戻し, その左辺と展開した式を一つの合同に入れる．例えば, 図 1 の $x1, x2, y1$ が属する合同の名前がそれぞれ $cong1, cong2, cong3$ とすると，図 7 の $z1$ と $p3$ の計算木は $\phi(cong2 + cong3, cong1 + cong3)$ になる．親ノードを参照する場合，参照した時点で親ノードの計算木を更に展開することを式に表現しないといけない．例えば図 6(a) と図 6(b) の計算木は式に戻すと $\phi(1, ((\lambda + 1), (\lambda + 2)))_\lambda\}$ になる． λ は親ノードを参照することを示し，その親ノードの木を表した式に添え字 λ を付け，参照点と対応するようにする．

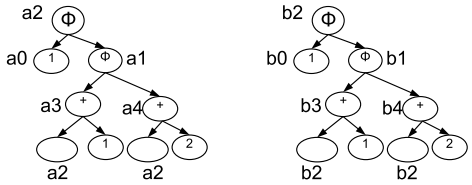
- 左辺が同じである合同を合併する．図 1 のプログラムでは，合同 $\{z1, p3, \phi(cong2 + cong3, cong1+cong3)\}$ が求められる．図 4 のプログラムでは合同 $\{a0, b0, 1\}$, $\{a2, b2, \phi(1, ((\lambda+1), (\lambda + 2)))_\lambda\}$, ... が求められる．

以上の手順で示したように，変数の振舞いを木構造で表現することによって，等価変数が求められる．



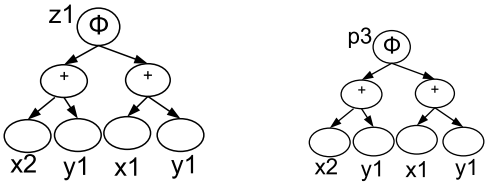
(a) 図 1 の $z1$ の計算木　(b) 図 4 の $a2$ の計算木

図 5　計算木



(a) 図 4 の $a2$ を展開した計算木　(b) 図 4 の $b2$ を展開した計算木

図 6　計算木の展開の例



(a) 図 1 の $z1$ を展開した計算木　(b) 図 1 の $p3$ を展開した計算木

図 7　ϕ 計算による計算木の展開

■命令文の変形の比較照合　命令文の変形は色々あるが，以下のように比較しながら，命令文の変形を認識する．

- 字面が完全に同じであれば同じ命令文とする
- 命令文の中，違った部分が同じ合同に属する命令文は同じ命令文とする．
- 条件付き JUMP 文が存在して，片方が無効になるとき，1 つの JUMP 文と見なす．
- 最適化の後無くなった命令文は削除した命令文とする．
- 左辺が同じであるが，右辺が異なる命令文は書き換えた命令文とする．

- 余計に生成された JUMP 文は無視する．
- …

■辺の変形の比較照合　命令文の認識が終ったら，辺の変形を認識する．

- 同じ命令文または書き換えた命令文だと認識されたノード同士を繋げた辺は同じ辺とする．
- 最適化によってなくなった辺は削除した辺とする．
- 最適化によって増えた辺は挿入した辺とする．
- …

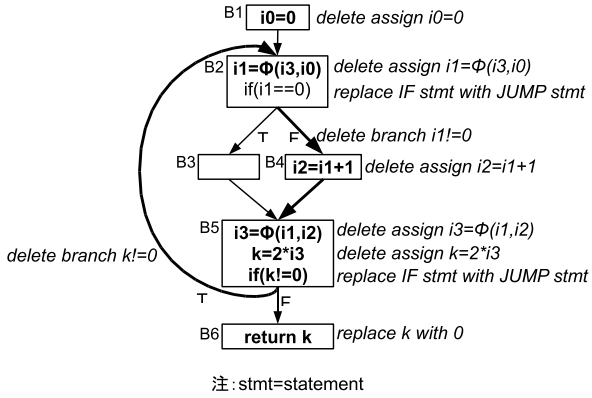


図 8　モデルの例

4.2.3　モデル検査

■モデルの生成　プログラムの解析及び比較をしたあと，最適化前後の制御フローグラフ及び以上の手順で命令文と辺を比較照合した情報に基づいてモデルを生成する．

最適化前の制御フローグラフを元にまずモデルを生成し，比較照合の情報によって変更なしの部分はそのままとし，最適化によって変更した部分はその内容をそのモデルの上に付け加える．

変更の内容によって，モデルの処理も色々ある．特に，削除した辺はモデルから辿れないようにする処理が必要である．

定義 1 (制御フローに基づいたモデル)
1 つの文をノードとする $G = (N, E)$ を考える．ただし，N はノードの集合，E はノード間の制御フローに基づいた有向辺の集合とする．

原始命題全体の集合を AP とし，ノード $n \in N$ に対し，そこで成り立つ原始命題の集合 $L(n)$ を与える写像を $L\colon N \to 2^{AP}$ とする．三つ組 $M = (N, E, L)$ を **モデル**ということにする．これは Kripke 構造であり，N が**状態**の集合，$E \subseteq N \times N$ は状態間の**遷移**に相当する．状態 n と n' の間に遷移がある，すなわち $(n, n') \in E$ であることを $n \to n'$ とも表記する．

定義 2 (モデル上の経路)
モデル $M = (N, E, L)$ において，状態の無限列 $n_0, n_1, n_2, ...$ が，任意の $i \geq 0$ について $n_i \to n_{i+1}$ であるとき，この無限列を**無限経路**という．また，状態の有限列 $n_0, n_1, ..., n_m$ が，任意の $i\,(0 \leq i < m)$ について $n_i \to n_{i+1}$ かつ $\forall n \in N, \neg(n_m \to n)$ であるとき，この有限列を**有限経路**という．無限経路と有限経路を合わせて**経路（パス）**という．

$n_1 \to n_2$ であるとき，この遷移関係の逆を**逆向きの遷移**といい，$n_2 \to^{\circ} n_1$ と表す．この逆向きの遷移に対して定義される経路を**逆向きの経路**という．

■最適化による変更をモデル上で反映　最適化による変更をモデル上で以下の通りに反映させる．

- 削除された命令文のノードをそのまま残し，削除されたことを原始命題で示す．
- 書き換えられた命令文のノードをそのまま残し，書き換えられた内容を原始命題で示す．
- 挿入した命令文はモデルに新しいノードを追加し，追加したことを原始命題で示す．
- 削除した辺は，この辺をモデルから取り除き，この辺を通るための条件文に辺を削除したことを原始命題で示す．
- …

図 8 は 2.2 節で説明したプログラムの例のモデルである．条件分岐を考慮した定数伝播により変更した部分は太字や太線で表示し，変更の原始命題を右側に記した．

5　定式化

最適化の変形が満たすべき条件を 3 節で述べた時相論理式で記述するのは定式化である．命令文の削

除，命令文の挿入，命令文の書き換え，分岐の削除の 4 種類の変形に対して，4 つの論理式を定式化した．

定式化の時，検査用の論理式について，詭弁論理（sophism) 式を書いてはいけない．例えば，2 つの条件が互い成り立つことを根拠とする**循環論証（circular reasoning）**や結論を導く事が出来る情報に論点が成り立つことを仮定した前提で導いた結論を含めている**論点先取（petitio principii）**を避けるべきである．

図 8 の例で $i0 = 0$ の代入文が削除できる条件は以下のように書く．

$\neg(E\ (\neg\ def(i0)\ U\ (use(i0) \wedge \neg(rm\_use(i0)))))$

この式は定義した後使用しないか、最適化により使用が削除されたかという条件を意味する．

図 8 の例で $i0 \neq 0$ の場合分岐する辺が削除できる条件は以下のように書く．

$\overleftarrow{A}\ (\ true\ U\ (\ condition(i1 \equiv 0, false)\ ))$

これは，この辺を除いてどんな経路からでも、この辺を通る条件を満たさないという条件を意味する．この辺が通れる条件をモデル上で検査するとき，モデルがこの辺も含んでいると，この辺を通れることを前提にして，通れるかどうかを検査することになり，論点先取になるため，モデルから検証の対象辺を取り除く．

その他，命令文 $x = e$ の挿入と命令文 $x = e$ のオペランド $v1$ が $v2$ に書き換えられた場合の条件式はそれぞれ

$\neg E\ (\ transp(x)\ U\ use(x))$

$\overleftarrow{A}\ (\ transp(v1) \wedge transp(v2)\ U\ equal(v1, v2))$

になる．

従来研究 [21] は，最適化器の振舞いを対象としていた．これは最適化器や最適化器のアルゴリズムに依存するため，条件式が数十個にも及ぶことがある．これに対して，本研究は，最適化器や最適化器のアルゴリズムに依存しないため，条件式は 4 つで収まり，極めて簡単である．また，ユーザが定式化を行う時，最適化器と最適化のアルゴリズムを理解する必要がないため，使いやすい．新しい最適化を追加したり，最適化器のアルゴリズムを変えたりしても新しく定式化しなくてよい．

6 実際の最適化器への適用

この節では，提案手法を COINS の最適化器に適用して行った実験を基に，本手法の有用性について考察する．COINS コンパイラ [6] のバックエンド上では，低水準中間表現 LIR[15] を対象とした多くの最適化が実装されており，特に SSA 形式上での最適化が充実している [20].

我々は，LIR を対象とする次の最適化器について本手法を適用し検査を行った．

- 無用命令除去
- 定数の畳み込み
- コピー伝播
- 共通部分式除去
- ループ不変式移動
- 条件分岐を考慮した定数伝播
- 質問伝播に基づく大域値番号付けと部分冗長除去

6.1 複雑な最適化器への適用

COINS SSA 最適化モジュールには，条件分岐を考慮した定数伝播 [23] や質問伝播に基づく大域値番号付けと部分冗長除去 [20] といった，非常に複雑な最適化が実装されている．これらの最適化器には，バグがある可能性がある．これらの最適化器を本手法により検証できた．

6.1.1 条件分岐を考慮した定数伝播 (CSTP) についての考察

条件分岐を考慮した定数伝播 [23] は SSA 形式上での強力な定数伝播アルゴリズムである．その正しさの証明は非常に有用である．

2.2 節の図 2 は，条件分岐を考慮した定数伝播を行ったプログラムの例である．**抽象実行（abstract interpretation）**によって，図 2(a) のグラフの入り口から順に辿ると，$i1$ は常に 0 となり，ブロック $B4$ は到達不能であることなどが分かる．

最適化後は図 2(b) のようになる．$i0$ や $i1$，k など，値が常に定数となることが分かる変数の使用が定数に置き換えられ，その変数への代入文が削除さ

れる．また，$B4$ のような到達不能ブロックの文やそこへの分岐が削除される．

条件分岐を考慮した定数伝播はデータフロー方程式では扱えない．データフロー方程式は，μ 計算 [9] と同等の計算力があるといわれている．したがって μ 計算より計算力の低い CTL-FV モデル検査をプログラムの解析に用いる Lacey らの手法ではこの問題に対応できない．

佐原ら [21] は条件付き定数伝播を証明したと主張しているが，「辺が削除できる条件はこの辺が指している命令文が削除されたこと，（削除した辺が指している）命令文が削除できる条件はこの命令文を指している辺が削除されたこと．」といった循環論証になっているため，検証できているように見えるが，実際はできていなく，実装も無限循環に陥る．

6.1.2 質問伝播に基づく大域値番号付けと部分冗長除去 (PREQP) についての考察

2.1 節に述べたように，SSA 形式は変数の使用に到達する定義が一意に決定でき，プログラムの最適化に有利な形式である．しかし，図 1(a) のプログラムは図 1(b) に変換した後，$z1 = x3 + y1$ は $p1 = x2 + y1$ と $p2 = x1 + y1$ によって計算された結果を冗長に再計算しているが，SSA 変換により，その情報が失われた．この冗長性を除去するアルゴリズムとして，質問伝播に基づく大域値番号付けと部分冗長除去 (PREQP)[20] がある．図 1(b) のプログラムに PREQP を適用すると，図 9 になる．PREQP のアルゴリズムは複雑で正しいという確証を得るのは難しい．それに対して本研究は 4.2.2 節に述べた方法により，$z1$ と $p3$ は等価変数であり，$z1$ が $p3$ の代入で正しいことを簡単に認識できる．

6.2 実験

この節では，提案手法を COINS の最適化器に適用して行った実験を基に，本手法の有用性について考察する．

6.2.1 未知の誤りや曖昧な変形の発見

実験は 670 本のテストコード集合と 9 本のベンチマークを対象とした．

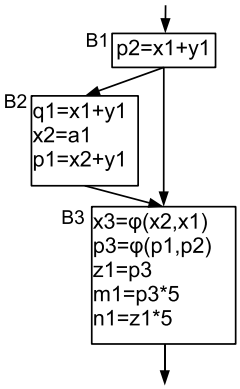


図 9　PREQP の例

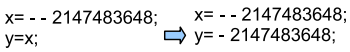


図 10　曖昧な変換による誤り

まだ実験中であるが，発見できるバグの種類と量について，これまでにない成果を上げられると予想される．また，検証時間もすべての従来研究より速いと予測した．具体的な実験データが完了し次第報告したい．以下は今の段階で発見したいくつの不具合の例である．

- 実装依存の変換.
  例 1：図 10 に示した変換について，$--2147483648$ が符号なしで -2147483648 が符号付の場合，同じビットパタンであるが，符号なしの変数を符号付の変数に代入するのは実装依存のため，誤りの可能性がある．
- 曖昧な変換.
  例 2：$x/8$ が最適化によって，$x >> 3$ になった．x が負数のとき C 言語では，$>>$ は実装依存であり，算術シフトの場合は大丈夫だが，論理シフトの場合は違った結果になる．
- 冗長を生じる変換.
  例 3：図 11 の変換は間違った変換はしないが，最適化の意図と逆に $temp = 2$ という冗長な代入が生じる．

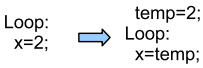


図 11　冗長な計算を生じる変換

6.2.2　不具合の報告

本システムは，最適化前後のプログラムに基づいてモデルを生成し，各変形箇所をその箇所の変形に応じた条件式によって検証する．その結果が偽の場合，その変形が条件を満たさないことを示すので，印字によって，エンドユーザに報告する．例えば，

$1:\ x = 1;$
$2:\ \ldots;$
$3:\ y = x + 1;$
$4:\ return\ y;$

の命令列に対して，最適化器が誤って $1:\ x = 1;$ を削除したとする．5 節に述べた命令文の削除が満たすべき条件式をこの変形に適用すると，以下のようになる．
$\neg(E\ (\neg\ def(x)\ U\ (use(x) \wedge \neg(rm\_use(x)))))$
$1:\ x = 1;$ が削除された後，$3:\ y = x + 1;$ によって使用されたため，条件式は $false$ になる．$false$ になった箇所の命令文，その命令文が属した関数の名前，行番号，誤りの変形の内容を以下のように印字する．

$FAULT\ in$ “$decode\_c$” $at\ 78\ line$：$x = 1;$
$the\ transformation\ is$：$delete(x = 1)$.
$the\ check\ formula\ is$：
$\neg(E\ (\neg\ def(x)\ U\ (use(x) \wedge \neg(rm\_use(x)))))$.

このように，最適化器の誤り箇所同定に役に立つような情報を提供する．

6.2.3　検査効率と検査効果

まだ実験中だが，ほかのバグも発見できると信じている．検査効率や効果について，実験データがまとまり次第報告したい．

7　関連研究

最適化の正しさに関する研究は多くなされている．この節では，本手法に関連の深いいくつかの研究と本手法の比較を述べる．

Lacey らは，時相論理 CTL-FV を用いて最適化のデータフロー解析を記述し，モデル検査により最適化を実行する手法を提案した [11, 12]．また，この手法により実現した最適化がプログラムの意味を保存するものであることを，いくつかの最適化について，簡単な方法で証明できることを示した．証明された最適化器は，常に正しく実行されることが保証され，バグのない最適化器といえる．しかし，データフロー方程式のみで記述できる，比較的簡単な最適化しか実現できないようである．例えば，6.1.1 節で説明した条件分岐を考慮した定数伝播は，我々の手法では扱えるが，Lacey らの手法では扱うことができない．

Lerner らは，時相論理を基にした独自のドメイン記述言語により最適化を記述し実行するシステムを提案した [13, 14]．これは，Lacey らの研究に似ているが，定理証明器を用いて最適化の正しさの証明をほぼ自動的に行えるという特徴がある．Lerner らのシステムも，Lacey らの手法と同様，複雑な最適化は扱えないようである．

Necula は，最適化前後のプログラムの意味が等しいことを，記号的に推測し評価することで検査する手法を提案した [17]．この検査手法は，最適化変換前後のプログラムに対して集合 $\{PC_s, PC_t, E\}$ を検査する．$\{PC_s, PC_t\}$ は最適化前後のプログラムにおいて，等価であるべき箇所 ($notation$) のペアである．E はその対応関係 ($equivalence\ criterion$) である．この研究は演算の強さの軽減 [8] や，レジスタ割当てまたはコードスケジューリングなどプログラムの変数や構造が変わる最適化を検証できる利点がある．しかし，この研究は等価関係のある規則に沿った推測であり，厳密なプログラムの意味の保存は保証されない．推測が失敗する場合があり，複雑な最適化については精度が低いと思われる．バグを発見した際の原因の特定は本手法ほど容易ではない．また，規則の基準を緩くすると，バグではないものもたくさん含んだ情報を出し，本当のバグを判断できなくなり，逆に規則を厳しくすると，バグであるものが

報告されないという矛盾がある．また，この手法は原理的には各々の最適化に依存しないが，ユーザにとって最適化器を理解し，最適化の種類によって規則の基準を調整する必要がある．

Rinard らは，最適化前後のプログラムの意味が等しくなるための変数の値の条件を割り出し，最適化の実行後に，プログラムの意味が保存されているかどうか証明する手法を提案した [18]．この手法は厳密なプログラムの意味の保存を保証できるようだが，具体的なアルゴリズムの記載がなく，どの程度実用的な手法なのか不明である．

佐々らの研究 [19] や Jaramillo らの研究 [10] は最適化前後のプログラムの実行を比較照合する．しかし，この研究はいくつのバグを見つけたが，トレース文がギガバイトのオーダになり，検証時間も何十分もかかる．また，理論上の裏づけが弱い．

佐原らの研究 [21] は最適化器の振舞いを検査の対象とし，最適化しながら最適化のアルゴリズムと関わる箇所にマークをつけ，それらのマークの関係を証明して検証する．しかし、この研究はいくつの欠点がある．まず，ユーザが最適化のアルゴリズムを理解し，アルゴリズムの対応する箇所を分析する必要がある．普通のユーザはコンパイラやコンパイラ最適化器，コンパイラ最適化器のアルゴリズムなどを理解し，正しく定式化するのに大変苦労すると思われる．次は，同じ最適化にしても，アルゴリズムが変わったら，適用できなくなり，新たに定式化する必要がある．また，アスペクト指向により最適化器に手を入れる必要がある．さらに，定式化の種類が多すぎる．質問伝播に基づく大域値番号付けと部分冗長除去（PREQP）や SSA 形式での部分冗長性除去（PRE）について表現できない．最後に，変形と関わる箇所の間の関係を検証するため，循環論証 に陥りやすい，実際にも分岐を考慮した定数伝播 [23] は無限循環に陥って検証できない．

8　まとめ

我々は，CTL-FV のモデル検査を利用して，最適化による変形がプログラムの意味を保存する等価変形であったかどうか検査する手法を提案した．提案した手法は，最適化前後のプログラムを比較分析し，最適化による変形がプログラムの意味を変えないために満たすべき性質を CTL-FV で記述しておき，モデル検査を行うことで性質を満たすことを調べる．最適化器や最適化のアルゴリズムに依存しないため，ユーザにとって使いやすい．

この手法により，様々な既存の最適化器の検査が行えた．また，提案手法を実装し実験したところ，最適化器の未知の不具合をいくつか発見することができた．まだ実験中であるため，今の段階では結論をつけられないが，高速で有効であると思われる．

9　今後の課題

今後の課題としては，主に次の 3 つが挙げられる．

■厳密な証明　本手法で記述した最適化の変形箇所の仕様は，それ単独で最適化の正しさを保証するものではない．多くのものは，直観的に明らかであったり，他の文献ですでに証明がなされていたりするが，厳密には，対象プログラムの意味論を正確に定義し，その上で「仕様を満たすならば正しい」という証明を与えるべきである．これは，定理証明器 Coq[7] などを用いるとよいと思われる．

■より複雑な最適化器の検証　COINS SSA 最適化モジュールには，変数の名前が変わる帰納変数の演算の強さの軽減と判定の置き換え [8] や，ループの構造を変えるループ展開 [16] などの最適化がある．これらは，今の段階では，最適化の検査は行えないが，今後，変数の等価性の評価を拡張することによって検証したい．

■実用化へ　本研究の手法は従来研究の欠点を克服し，効果が良く，高速で使いやすいなどの特徴があるため，実用化に踏み出したい．

謝辞

本研究の一部は科学研究費補助金の補助を受けた．

参考文献

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D.:*Compilers: Principles, Techniques, and Tools 2nd ed.*, Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Andrew W. Appel and Jens Palsberg: Modern Compiler Implementation in Java 2nd ed., Cam-

bridge University Press, 2002.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck: Detecting equality of variables in programs, *In Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 1–11, 1988.

[4] E. M. Clarke, E. A. Emerson and A. P. Sistla: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, Vol. 17, No. 3, pp. 244–263, 1986.

[5] Clarke, E. M. Jr., Grumberg, O. and Reled, D. A.: Model Checking. The MIT Press, 1999.

[6] COINS-Project：COINS Home Page. http://www.coins-project.org

[7] Coq Home Page. http://coq.inria.fr/

[8] Keith D. Cooper, L. Taylor Simpson and Christopher A. Vick: Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, Vol. 23, No. 5, pp. 603–625, 2001.

[9] Van Leeuwen, J. (Ed.): Handbook of Theoretical Computer Science Vol. B, Formal Models and Semantics, Elseveir Science Publishers B. V., 1990. (広瀬ほか訳，丸善)

[10] Jaramillo, C., Gupta, R. and Soffa, M. L.: Debugging and Testing Optimizers through Comparison Checking, *Electronic Notes in Theoretical Computer Sciences*, Vol. 65, No. 2, pp. 1–17, 2002.

[11] Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Proving correctness of compiler optimizations by temporal logic, *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 283–294, 2002.

[12] Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation*, Vol. 17, No. 3, pp. 173–206, 2004.

[13] Sorin Lerner, Todd Millstein and Craig Chambers: Automatically proving the correctness of compiler optimizations, *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 220–231, 2003.

[14] Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformation via local rules, *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, ACM Press, pp. 364–377, 2005.

[15] COINS-Project：COINS プロジェクト LIR 仕様書 http://www.coins-project.org/spec/lir.pdf

[16] 中田育男：コンパイラの構成と最適化．朝倉書店，1999.

[17] Necula, G. C.: Translation validation for an optimizing compiler, PLDI ' 00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, ACM Press, 2000, pp. 83-94.

[18] Rinard, M. and Marinov, D.: Credible compilation with pointers, Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy, Jul. 1999.

[19] Sassa, M. and Sudo. D.: Experience in Testing Compiler Optimizers Using Comparison Checking, *2006 International Conference on Programming Languages and Compilers (PLC '06)*, Vol. II, pp. 837-843, June 2006

[20] 佐々研究室: 静的単一代入形式最適化システム外部仕様書, 2006. http://www.is.titech.ac.jp/˜sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf.

[21] 佐原聡一郎，佐々政孝: 時相論理を用いたコンパイラ最適化器の実行の正しさの検査，コンピュータソフトウェア, pp. 151–166, 2008.

[22] 佐々 政孝：プログラミング言語処理系．岩波書店，1989.

[23] Wegman, M.N. and Zadeck, F.K.: Constant propagation with conditional branches, *ACM Trans. Prog. Lang. Syst.,*Vol. 13, No. 2, pp. 181–210, 1991.